

Learning efficient logic programs

Andrew Cropper¹ · Stephen H. Muggleton²

Received: 17 November 2017 / Accepted: 5 April 2018 / Published online: 23 April 2018
© The Author(s) 2018

Abstract When machine learning programs from data, we ideally want to learn efficient rather than inefficient programs. However, existing inductive logic programming (ILP) techniques cannot distinguish between the efficiencies of programs, such as permutation sort ($n!$) and merge sort $O(n \log n)$. To address this limitation, we introduce Metaopt, an ILP system which iteratively learns lower cost logic programs, each time further restricting the hypothesis space. We prove that given sufficiently large numbers of examples, Metaopt converges on minimal cost programs, and our experiments show that in practice only small numbers of examples are needed. To learn minimal time-complexity programs, including non-deterministic programs, we introduce a cost function called *tree cost* which measures the size of the SLD-tree searched when a program is given a goal. Our experiments on programming puzzles, robot strategies, and real-world string transformation problems show that Metaopt learns minimal cost programs. To our knowledge, Metaopt is the first machine learning approach that, given sufficient numbers of training examples, is guaranteed to learn minimal cost logic programs, including minimal time-complexity programs.

1 Introduction

Suppose you want to machine learn a program from the following examples:

```
f([p,r,o,g,r,a,m],r).  
f([i,n,d,u,c,t,i,o,n],i).
```

Editors: Fabrizio Riguzzi, Nicolas Lachiche, Christel Vrain, Riccardo Zese and Elena Bellodi.

✉ Andrew Cropper
andrew.cropper@cs.ox.ac.uk
Stephen H. Muggleton
s.muggleton@imperial.ac.uk

¹ Department of Computer Science, University of Oxford, Oxford, UK

² Department of Computing, Imperial College London, London, UK

These examples are described as Prolog facts. The first argument is the input list. The second argument is the output, and is the first duplicate element in the input. Given these examples and the background predicates shown in Fig. 1, Metagol (Muggleton et al. 2015; Cropper and Muggleton 2016a, b), a state-of-the-art inductive logic programming (ILP) system, learns the program shown in Fig. 2a. This recursive program goes through the elements of the list checking whether the same element exists in the rest of the list, i.e. finds the first duplicate element. By contrast, consider the program shown in Fig. 2b, which was learned by Metaopt, a new ILP system introduced in Sect. 4. This program first sorts the list and then goes through checking whether any adjacent elements are the same. Although larger, both in terms of clauses and literals, the program learned by Metaopt is more efficient ($O(n \log n)$) than the program learned by Metagol ($O(n^2)$). This efficiency difference is shown in Fig. 2c, which compares the running times of the two programs on randomly generated examples.

As this *find duplicate* scenario shows, when machine learning programs from examples, we should consider the efficiency of learned programs. However, existing ILP systems cannot distinguish between the efficiencies of programs, and typically rely on an Occamist bias to learn textually simple programs, such as those using the fewest literals (Law et al. 2014) or fewest clauses (Muggleton et al. 2015).

A recent paper (Cropper and Muggleton 2015) attempted to address this issue by introducing *Metagol_O*, an ILP system based on meta-interpretive learning (MIL) (Muggleton et al. 2014, 2015; Cropper and Muggleton 2016a). *Metagol_O* learns minimal *resource complexity* robot strategies, described as dyadic logic programs, where resource complexity is the sum of the action costs required to execute a strategy.

In this paper, we introduce Metaopt, which generalises *Metagol_O* by adding a general cost function into a meta-interpretive learner, where specific cost functions are provided as

Fig. 1 Background knowledge for the *find duplicate* problem. The definition for *mergesort/2* is omitted for brevity

```

head([H|_],H).
tail(_|T],T).
element([X|_],X):-!.
element(_|T],X):-element(T,X).
mergesort(A,B):- ...
    
```

```

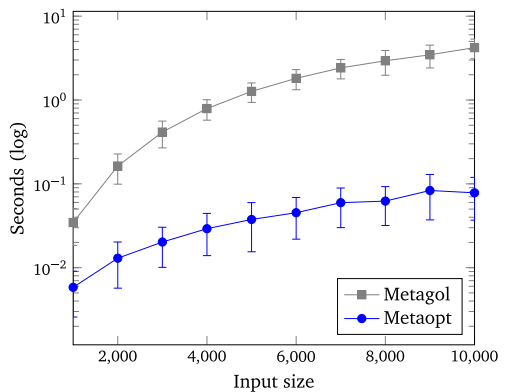
f(A,B):-head(A,B),f_1(A,B).
f(A,B):-tail(A,C),f(C,B).
f_1(A,B):-tail(A,C),element(C,B).
    
```

(a)

```

f(A,B):-mergesort(A,C),f_1(C,B).
f_1(A,B):-head(A,B),f_2(A,B).
f_1(A,B):-tail(A,C),f_1(C,B).
f_2(A,B):-tail(A,C),head(C,B).
    
```

(b)



(c)

Fig. 2 a An inefficient program learned by Metagol. b An efficient program learned by Metaopt. The predicates *f1/2* and *f2/2* are invented. c The running times of the two programs on randomly generated examples

background knowledge. Metaopt uses a search procedure called *iterative descent*, introduced in Cropper and Muggleton (2015) but generalised in this paper, to iteratively learn lower cost programs, each time further restricting the hypothesis space. We prove that given sufficiently large numbers of examples, Metaopt converges on minimal cost programs, and our experiments show that in practice only small numbers of examples are necessary. In contrast to *Metagol_O*, Metaopt is not restricted to dyadic logic programs, does not need designated input-output arguments, can handle negative examples, and considers backtracking steps when measuring program costs, crucial when learning non-deterministic logic programs. To learn minimal time-complexity programs, we introduce a cost function called *tree cost*, which measures the size of the SLD-tree searched when a program is given a goal. Our experiments on programming puzzles, robot strategies, and real-world string transformation problems show that Metaopt learns minimal tree cost programs.

Our specific contributions are as follows:

- We describe a general framework for learning minimal cost logic programs (Sect. 3).
- We extend MIL to support learning minimal cost logic programs (Sect. 3).
- To learn minimal time-complexity logic programs, we introduce a cost function called *tree cost* which is based on SLD-tree sizes (Sect. 3).
- We introduce Metaopt, a MIL implementation, and prove that it converges on minimal cost programs given sufficiently large numbers of examples (Sect. 4).
- We show through experimentation that Metaopt converges on minimal cost programs given small numbers of examples (Sect. 5).
- We demonstrate the generality of Metaopt by simulating *Metagol_O* to learn efficient robot strategies (Sect. 5).
- We show that Metaopt learns more efficient programs than existing ILP systems on real-world string transformation problems (Sect. 5).

2 Related work

Universal search If nothing is known about a problem besides input/output examples, and assuming that the solution can be verified in polynomial time, Levin’s universal search (Levin 1984) is the asymptotically fastest way of finding a program to solve the problem. Levin search differs from our work because it returns the first (and smallest) program that solves a problem, which is not necessarily the most efficient program. By contrast, after finding a program, our approach, Metaopt, continues to search for more efficient programs using the cost of the previously found program to restrict the hypothesis space. In addition, for many problems, it is unlikely that the most efficient program can be encoded with a small number of bits, making Levin search impractical. By contrast, although our approach is less general, because it assumes background knowledge of the problem, it is more practical.

Deductive program synthesis In contrast to universal search methods, deductive program synthesis (Manna and Waldinger 1979) approaches build programs from full specifications, where a specification precisely states the requirements and behaviour of the desired program. Both Kant (1983) and Zelle and Mooney (1993) synthesise efficient programs from complete specifications. In Zelle and Mooney (1993) the authors take as input a program to sort lists and use an explanation-based learning (Mitchell 1997) approach to speed-up the execution of the program by analysing program execution traces over training examples. A similar, yet distinct, approach is that of logic program transformation (Petrossi

and Proietti 1994). This approach starts with an initial program and successively applies transformation rules to programs to improve on previous versions. The aim is to end up with final program that has the same meaning as the initial one, but is preferably better in some way, such as being more compact or more efficient (Pettorossi and Proietti 1994). In contrast to these approaches, our approach induces programs from incomplete specifications in the form of input/output examples. In addition, the aforementioned approaches do not guarantee that the resulting program is optimal in terms of efficiency. By contrast, we prove that Metaopt finds the minimal cost program given sufficiently large numbers of examples (Theorem 1), and Experiment 1 shows that in practice only small numbers of examples are necessary.

Program induction Program induction, also known as inductive programming (Gulwani et al. 2015), refers to inducing programs from incomplete specifications, typically input/output examples. Early work includes Plotkin on least generalisation (Plotkin 1969, 1971), Vere on induction algorithms for expressions in predicate calculus (Vera 1975), and Summers on inducing Lisp programs (Summers 1977). Interest in program induction has grown recently, partially due to the success of mass-market tools, such as FlashFill (Gulwani 2011). Most forms of program induction are biased towards learning simple programs, typically those with minimal textual complexity, such as the number of clauses (Muggleton et al. 2015) or the number of literals (Law et al. 2014). This bias ignores the efficiency of hypothesised programs. In ILP, for instance, Golem (Muggleton and Feng 1990) and Progol (Muggleton 1995) can both learn sorting algorithms from examples, but when given background predicates *partition/3* and *append/3*, suitable for learning quick sort, both systems learn variants of insertion sort, because the definition is smaller and there is no bias for learning more efficient algorithms. By contrast, Metaopt is biased towards learning efficient programs.

AI planning In AI, planning typically involves deriving a sequence of actions to achieve a specific goal from an initial situation (Russell and Norvig 2010). Planning research mostly focuses on efficiently learning plans (Hoffmann and Nebel 2001). However, we are often interested in plans that are optimal with respect to an objective function which measures the quality of a plan. A common objective function is the length of the plan (Xing et al. 2006), and existing systems can learn optimal plans based on this function (Eiter et al. 2003).

Plan length alone is only one criterion. If executing actions is costly, we may prefer a plan which minimises the overall cost of the actions, e.g. to minimise the use of resources. The answer set programming literature has started to address learning optimal plans by incorporating action costs into the learning (Eiter et al. 2003; Yang et al. 2014). In contrast to these approaches, in Sect. 5.4, we use Metaopt to learn robot *strategies* (Cropper and Muggleton 2015), representing a (sometimes infinite) set of plans, which contain conditions and recursion.

Various machine learning approaches support constructing strategies, such as the SOAR architecture (Laird 2008), reinforcement learning (Sutton and Barto 1998), and action learning in ILP (Moyle and Muggleton 1997; Otero 2005). Relational markov decision processes (van Otterlo and Wiering 2012) provide a general setting for reinforcement learning. Strategies can be viewed as a deterministic special case of markov decision processes (MDPs) (Puterman 2014). Unlike these approaches, we learn recursive logic programs, including the use of predicate invention for automatic problem decomposition.

Efficient logic programs Kaplan (1988) describes a method for estimating the average-case complexity of deterministic logic programs. However, in contrast to functional and imperative programs, logic programs can be non-deterministic, i.e. a logic program may return multiple solutions. Multiple solutions to a logic program are found by searching a SLD-tree for a SLD-

refutation, and then backtracking to find other SLD-refutations. Debray et al. (1997) introduce a semi-automatic method to estimate the worst-case time complexity of deterministic and non-deterministic logic programs. However, their approach requires meta-information, such as mode declarations and type information. In contrast to these approaches, we introduce a cost function which estimates the worst-case complexity of both deterministic and non-deterministic logic programs. Our approach does not require meta-information, and works by measuring the size of the SLD-tree searched to find a SLD-refutation of a goal. In addition, the aforementioned approaches do not consider how to machine learn efficient programs.

In Sect. 4, we introduce Metaopt, which is used in Experiments 1, 2, and 3 to learn minimal time-complexity programs by iteratively restricting the hypothesis space by bounding the number of resolutions allowed to find a hypothesis. Our approach is similar to one proposed by Blum and Blum (1975) which was later adapted by Shapiro (1983) who uses the notion of *h-easy* functions to limit the search for a hypothesis, where an atom A is *h-easy* with respect to a logic program P if there exists a derivation of A from P using at most h resolution steps. Shapiro's approach measures the number of resolutions in the derivation of A from P , and thus ignores backtracking steps. By contrast, our approach uses the notion of *tree cost* to measure the total number of resolutions required to find a SLD-refutation of a goal with respect to a program, i.e. tree cost includes backtracking steps.

Our approach of evaluating a hypothesis based on the SLD-tree size is similar the the idea of Muggleton et al. (1992), who combined proof complexity (the number of choice points in a SLD-refutation) with other factors, such as hypothesis length and predictive accuracy, to measure the *significance* of a hypothesis. However, the authors only considered how to characterise the significance of a hypothesis. By contrast, we use a meta-interpretive learner to simultaneously induce and evaluate the time complexity of a hypothesis.

Metagol₀ To our knowledge, the only work on inducing efficient programs is the work of Cropper and Muggleton (2015), who introduce an ILP system called *Metagol₀* and show that it learns efficient robot strategies. A robot strategy is a dyadic logic program where the first argument of each predicate is the input and the second argument is the output. Each argument is a state description, represented as a list of Prolog atoms. Each predicate represents an action which modifies the state. The resource complexity of a strategy is maintained in the state as a monadic atom named *energy*. Each time a robot action is successfully executed, the resource complexity is increased by an amount specified by the user. *Metagol₀* is not a general approach for learning efficient logic programs and has several limitations. Problems must be represented as dyadic programs, which is inconvenient and may lead to reduced learning performance because concepts may be less succinctly represented. Also, the user must specify predicate costs. For instance, if *mergesort/2* is part of the background knowledge, then the user must specify its cost. If no costs are given, *Metagol₀* assumes uniform costs, and cannot, for instance, distinguish between *mergesort/2* and *tail/2*. Finally, *Metagol₀* ignores failed actions and cannot accurately measure the time complexity of programs with backtracking. Our system, Metaopt, addresses all of these issues.

3 Framework

In this section, we describe the *cost minimisation problem*. We also describe MIL, which we extend to support learning minimal cost programs. We assume familiarity with logic programming (Nienhuys-Cheng and Wolf 1997).

3.1 Cost minimisation problem

We assume a language of examples \mathcal{E} , background knowledge \mathcal{B} , and hypotheses \mathcal{H} . We denote the Herbrand base of the language L as σ_L . We denote the power set of the set S as 2^S .

We first define a cost function that measures the cost of a program with respect to an atom:

Definition 1 (*Program cost*) A program cost function is of the form:

$$\Phi : \mathcal{H} \times \sigma_{\mathcal{B} \cup \mathcal{E}} \rightarrow \mathbf{N}$$

A program cost function forms part of the cost minimisation input:

Definition 2 (*Cost minimisation input*) The cost minimisation input is a triple (B, E, Φ) where:

- $B \subseteq \mathcal{B}$ is background knowledge
- $E = (E^+, E^-)$ is a pair where $E^+ \subseteq \mathcal{E}$, $E^- \subseteq \mathcal{E}$ are sets of atoms representing positive and negative examples respectively
- Φ is a program cost function

We measure the maximum (i.e. worst-case) cost of a program over a set of examples:

Definition 3 (*Maximum cost*) Let (B, E, Φ) be a cost minimisation input, where $E = (E^+, E^-)$. Then the maximum cost of a program $H \in \mathcal{H}$ is:

$$\text{max_cost}(\Phi, H, (E^+, E^-)) = \text{max}_{e \in E^+ \cup E^-} \Phi(H, e)$$

We also define a function that measures the size of a logic program:

Definition 4 (*Program size*) The size $\text{size}(H)$ of the program H is the number of clauses in H .

We use the maximum cost and size of a program to define an efficiency ordering over programs:

Definition 5 (*Efficiency ordering \preceq_Φ*) Let (B, E, Φ) be a cost minimisation input and $H_1, H_2 \in \mathcal{H}$. Then $H_1 \preceq_\Phi H_2$ iff either:

1. $\text{max_cost}(\Phi, H_1, E) < \text{max_cost}(\Phi, H_2, E)$
2. $\text{max_cost}(\Phi, H_1, E) = \text{max_cost}(\Phi, H_2, E)$ and $\text{size}(H_1) \leq \text{size}(H_2)$

This ordering priorities programs first by their maximum cost, then by their size. We use this ordering to define the cost minimisation problem. For convenience, we first define the version space (Mitchell 1997), which contains only hypotheses consistent with the examples:

Definition 6 (*Version space*) The version space $\mathcal{V}_{B,E}$ of a cost minimisation input (B, E, Φ) contains the hypotheses consistent with E :

$$\mathcal{V}_{B,E} = \{H \in \mathcal{H} \mid B \cup H \models E^+, B \cup H \not\models E^-\}$$

We now define the cost minimisation problem:

Definition 7 (*Cost minimisation problem*) Given a cost minimisation input (B, E, Φ) , the cost minimisation problem is to return a program $H \in \mathcal{V}_{B,E}$ such that $H \preceq_\Phi H'$ for all $H' \in \mathcal{V}_{B,E}$.

Table 1 Example metarules. The letters P , Q , R , and F denote existentially quantified variables. The letters A , B , and C denote universally quantified variables

Name	Metarule
Ident	$P(A, B) \leftarrow Q(A, B)$
Precon	$P(A, B) \leftarrow Q(A), R(A, B)$
Curry	$P(A, B) \leftarrow Q(A, B, F)$
Chain	$P(A, B) \leftarrow Q(A, C), R(C, B)$
Tailrec	$P(A, B) \leftarrow Q(A, C), P(C, B)$

3.2 Meta-interpretive learning

We now extend MIL to support the cost minimisation problem. MIL is a form of ILP based on a Prolog meta-interpreter. The key difference between a MIL learner and a standard Prolog meta-interpreter is that whereas a standard Prolog meta-interpreter attempts to prove a goal by repeatedly fetching first-order clauses whose heads unify with a given goal, a MIL learner additionally attempts to prove a goal by fetching higher-order metarules (Table 1), supplied as background knowledge, whose heads unify with the goal. The resulting meta-substitutions are saved and can be reused in later proofs. Following the proof of a set of goals, a logic program is formed by projecting the meta-substitutions onto their corresponding metarules. A standard MIL input is defined as:

Definition 8 (*MIL input*) A MIL input is a pair (B, E) where:

- $B = B_C \cup M$ where B_C is a set of definite clauses and M is a set of metarules
- $E = (E^+, E^-)$ is a pair where E^+ and E^- are sets of atoms representing positive and negative examples respectively

A standard MIL learner is defined as:

Definition 9 (*MIL learner*) Given a MIL input (B, E) , a MIL learner returns a program $H \in \mathcal{V}_{B,E}$.

We extend MIL to support the cost minimisation problem. We first extend the MIL input:

Definition 10 (*Cost minimal MIL input*) A cost minimal MIL input is a triple (B, E, Φ) where B and E are as in a standard MIL input and Φ is a program cost function.

We now define a cost minimal MIL learner:

Definition 11 (*Cost minimal MIL learner*) Given a cost minimal MIL input (B, E, Φ) , a cost-minimal MIL learner returns a program $H \in \mathcal{V}_{B,E}$ such that $H \preceq_{\Phi} H'$ for all $H' \in \mathcal{V}_{B,E}$.

In Sect. 4, we introduce Metaopt, a MIL learner that solves the MIL cost minimisation problem.

3.3 Tree cost minimisation

The cost minimisation input includes a program cost function (Definition 1). We now introduce a cost function for learning minimal time-complexity logic programs. In computer science, time complexity refers to the time an algorithm needs to perform some computation. In logic programming, computation is formalised by means of SLD-resolution. Given a logic

program H and a goal G , computation involves finding a SLD-refutation of $H \cup \{G\}$. A SLD-refutation is found by searching a SLD-tree, which contains all possible SLD-derivations, and thus all possible SLD-refutations. Prolog searches for SLD-refutations using a depth-first search (Sterling and Shapiro 1994). We can therefore measure the runtime (time complexity) of a Prolog program as a function of the size of the SLD-tree that is being searched. For a positive example, we can measure the size of the leftmost branch of the SLD-tree in which the first SLD-refutation is found, i.e. the leftmost successful branch:

Definition 12 (*Successful branch*) Let H be a definite program, G be an initial goal, and T be a SLD-tree for $H \cup \{G\}$. Then a successful branch is a path between the root (G) and a leaf containing the empty clause.

We measure the size of the leftmost successful branch:

Definition 13 (*Branch size*) Let H be a definite program, G a goal, T a SLD-tree for $H \cup \{G\}$, and L be the leftmost successful branch of T . Then the branch size $branch_size(H, G)$ is the number of resolutions prior to and including L in the depth-first enumeration of T .

For a negative example, we can measure the size of the finitely failed SLD-tree:

Definition 14 (*Finitely failed tree*) Let H be a definite program, G be an initial goal. Then a finitely failed SLD-tree for $H \cup \{G\}$ is one which is finite and contains no successful branches.

We measure the size of the finitely failed SLD-tree:

Definition 15 (*Failed tree size*) Let H be a definite program, G be an initial goal, and T be a finitely failed SLD-tree for $H \cup \{G\}$. Then the failed tree size $tree_size(H, G)$ is the number of resolutions in the depth-first enumeration of T .

We now define our *tree cost* function:

Definition 16 (*Tree cost*) Let H be a definite program, G a goal, and T be a SLD-tree for $H \cup \{G\}$. Then the tree cost $tree_cost(H, G)$ is:

$$tree_cost(H, G) = \begin{cases} branch_size(H, G) & \text{if } T \text{ has a successful branch} \\ tree_size(H, G) & \text{if } T \text{ is finitely failed} \end{cases}$$

In Experiments 1, 2, and 3, Metaopt uses tree cost to learn minimal time-complexity programs.

4 Implementation

In this section, we introduce Metaopt, a MIL implementation that learns minimal cost logic programs. We also introduce two cost function implementations for learning minimal tree cost (Definition 16) and minimal resource complexity (Cropper and Muggleton 2015) programs. Finally, we describe $Metaopt_O$ which is used as a comparator in the experiments in Sect. 5.

4.1 Metaopt

Metaopt extends $Metaopt_O$ (Cropper and Muggleton 2016b), an existing MIL implementation, to support learning minimal cost logic programs. The two key extensions are (1) the addition of a general cost function into the meta-interpreter, and (2) the use of a procedure called iterative descent to search for lower cost programs. We describe these two extensions in turn.

4.1.1 Meta-interpreter

The key extension in Metaopt is the addition of a proof cost, denoted by the variables C_i , into the meta-interpreter¹:

```
learn (Pos , Neg , Prog) :-
    prove (Pos , [] , Prog , 0 , _) ,
    \+ prove (Neg , Prog , Prog , 0 , _) .
prove ([] , Prog , Prog , C , C) .
prove ([Atom | Atoms] , Prog1 , Prog2 , C1 , C2) :-
    prove_aux (Atom , Prog1 , Prog3 , C1 , C3) ,
    prove (Atoms , Prog3 , Prog2 , C3 , C2) .
prove_aux (Atom , Prog , Prog , C1 , C2) :-
    pos_program_cost (Atom , Cost) ,
    C2 is C1+Cost ,
    get_max_cost (MaxCost) ,
    C2 < MaxCost .
prove_aux (Atom , Prog1 , Prog2 , C1 , C2) :-
    member (sub (Name , Subs) , Prog1) ,
    metarule (Name , Subs , (Atom :- Body)) ,
    prove (Body , Prog1 , Prog2 , C1 , C2) .
prove_aux (Atom , Prog1 , Prog2 , C1 , C2) :-
    metarule (Name , Subs , (Atom :- Body)) ,
    prove (Body , [sub (Name , Subs) | Prog1] , Prog2 , C1 , C2) .
```

The meta-interpreter works as follows. Given sets of atoms representing positive (Pos) and negative (Neg) examples, Metaopt tries to prove each positive atom in turn. Metaopt first tries to deductively prove an atom by calling `pos_program_cost/2`, which is defined as background knowledge. When an atom is proven this way, the cost of proving that atom is added to the overall proof cost. If the overall proof cost exceeds a bound (MaxCost), then the proof is terminated, as to ignore inefficient programs. This bound is determined by the iterative descent procedure, described in the next section. If Metaopt cannot deductively prove an atom, it tries to unify the atom with the head of a metarule (`metarule (Name, Subs, (Atom :- Body))`) and to bind the existentially quantified variables in a metarule to symbols in the predicate signature. Metaopt saves the resulting meta-substitutions, which are eventually used to form a program. Metaopt then tries to prove the body of the metarule recursively through meta-interpretation. After proving all positive atoms, a logic program is formed by projecting the meta-substitutions onto their corresponding metarules. Metaopt checks the consistency of the learned program with the negative examples. If the program is inconsistent, then Metaopt backtracks to explore different branches of the SLD-tree.

4.1.2 Iterative descent

The Metaopt meta-interpreter is controlled by the iterative descent algorithm:

```
metaopt (Pos , Neg) :-
    learn (Pos , Neg , Prog) ,
    max_program_cost (Prog , Pos , Neg , Cost) ,
```

¹ To aid readability, the Prolog code for the meta-interpreter is a slimmed down version of the one used in the experiments.

```

is_better(Cost),
set_max_cost(Cost),
set_best_program(Prog),
false.
metaopt(_,_) :-
get_best_program(Prog),
pprint(Prog).
max_program_cost(Prog, Pos, Neg, MaxCost) :-
assert_program(Prog),
findall(C, (member(Atom, Pos), pos_program_cost(Atom, C)), PosCosts),
findall(C, (member(Atom, Neg), neg_program_cost(Atom, C)), NegCosts),
append(PosCosts, NegCosts, Costs),
max_list(Costs, MaxCost),
retract_program(Prog).

```

Iterative descent works as follows. Starting at iteration 1, Metaopt uses iterative deepening on the number of clauses to find a consistent program H_1 with the minimal program size (Definition 4). The program H_1 is the quickest to learn because the hypothesis space is exponential in the number of clauses (Lin et al. 2014; Cropper and Muggleton 2016a). Metaopt then calculates the cost of H_1 using the `max_program_cost/4` predicate, which measures the maximum cost of proving the positive examples and disproving the negative examples. This cost becomes the maximum cost for the next iteration for both the meta-interpreter and the iterative descent algorithm. In iteration $i > 1$, Metaopt searches for a program H_i , again with minimal program size, but ensuring that the cost of H_i is less than the cost of H_{i-1} . Iterative descent continues until it cannot find a lower cost program.

We now prove that Metaopt solves the cost minimisation problem (Definition 7), i.e. that Metaopt converges on minimal cost programs given sufficiently large numbers of examples:

Theorem 1 (Metaopt convergence) *Assume E consists of m examples drawn from an enumeration of the infinite example space E' . Without loss of generality consider the hypothesis space formed of two programs H_1 and H_2 such that $H_1 \preceq_{\Phi} H_2$ for an arbitrary cost function Φ . Then for a sufficiently large value m , Metaopt will return H_1 in preference to H_2 .*

Proof Assume false, which, because of Definition 5, implies that one of the following conditions must hold:

1. $\max_cost(\Phi, H_1, E) > \max_cost(\Phi, H_2, E)$
2. $\max_cost(\Phi, H_1, E) = \max_cost(\Phi, H_2, E)$ and $size(H_1) > size(H_2)$

We consider these two cases in turn:

Case 1 With sufficiently large m there will exist an example e such that $\Phi(H_1, e) < \Phi(H_2, e)$ and $\Phi(H_2, e) > \Phi(H_2, e')$ for all other e' in E and $\Phi(H_1, e) > \Phi(H_1, e')$ for all other e' in E . In this case $\max_cost(\Phi, H_1, E) < \max_cost(\Phi, H_2, E)$ and Metaopt returns H_1 which contradicts the assumption, so we discard this case.

Case 2 Metaopt performs iterative deepening search (IDS) on the number of clauses. From the optimality of IDS, Metaopt returns H_1 which contradicts the assumption, so we discard this case.

These two cases are exhaustive, thus the proof is complete. \square

4.2 Program costs

Metaopt assumes a program cost function (Definition 1) as background knowledge. We now describe two cost function implementations.

Fig. 3 Implementation of a program cost function to measure tree cost

```
pos_program_cost(Atom, Cost):-
    statistics(inferences, I1),
    call(Atom),
    statistics(inferences, I2),
    Cost is I2-I1.

neg_program_cost(Atom, Cost):-
    statistics(inferences, I1),
    (call(Atom) -> true; true),
    statistics(inferences, I2),
    % subtract one to account for the
    % cost of the ifthen statement
    Cost is I2-I1-1.
```

Fig. 4 Implementation of a program cost function to measure resource complexity

```
pos_program_cost(Atom, Cost):-
    Atom=..[P,A,B],
    member(energy(E1), A),
    call(Atom),
    member(energy(E2), B),
    Cost is E2-E1.
neg_program_cost(_, _):-
    false.
```

4.2.1 Tree cost

Figure 3 shows the implementation of the tree cost functions (Definition 16) for positive and negative examples. This implementation uses an inbuilt feature of SWI-Prolog (Wielemaker et al. 2012) to measure the number of logical inferences needed to prove an atom, where an inference is defined as a call or redo on a predicate.² This approach measures backtracking steps, costs associated with non-dyadic predicates, and costs associated with trying to prove negative examples, none of which are supported by Metagol_O. In Experiments 1, 2, and 3, Metaopt uses this implementation to learn minimal tree cost programs, and thus minimal time-complexity programs.

4.2.2 Resource complexity

In Experiment 4, we reproduce an experiment from Cropper and Muggleton (2015) to show that Metaopt can learn minimal resource complexity robot strategies and thus subsumes Metagol_O. As explained in Sect. 2, Metagol_O maintains the resource complexity of a strategy in the state description as a Prolog fact called *energy*. Figure 4 shows the implementation of a function which accesses this value so that it can be used by Metaopt.

4.3 Metagol_O

In Experiments 2 and 3 we compare Metaopt with Metagol_O. However, we cannot use the implementation from Cropper and Muggleton (2015) because Metagol_O requires that prob-

² <http://www.swi-prolog.org/pldoc/man?predicate=statistics/2>.

lems be represented as dyadic robot strategies (as detailed in Sect. 2). Therefore, we simulate Metagol_O in Metaopt by defining a program cost predicate in which all dyadic predicates have a uniform cost of 1, which is the assumption in Metagol_O , and non-dyadic predicates have a cost of 0, because Metagol_O does not take these into account when calculating resource complexity.

5 Experiments

We now describe four experiments³ which test whether Metaopt learns minimal cost programs. We also compare Metaopt with Metagol and Metagol_O , which minimise textual complexity and resource complexity respectively.

5.1 Experiment 1: convergence on minimal cost programs

This experiment revisits the *find duplicate* problem from Sect. 1, where we are trying to learn a program to find a duplicate in a list. The aim is to test the claim that Metaopt converges on minimal cost programs given sufficient examples (Theorem 1). In particular, we want to see how many examples are required in practice to converge on a minimal cost program. We test the null hypothesis:

Null hypothesis 1 Metaopt cannot learn minimal cost programs without very large numbers of examples.

This experiment focuses on learning minimal tree cost programs, and thus minimal time-complexity programs.

Materials To refute null hypothesis 1, we must identify a minimal tree cost program in the hypothesis space, otherwise our hypothesis would be untestable. We provide Metaopt with background knowledge containing the *ident*, *chain*, and *tailrec* metarules (Table 1) and four predicates: *mergesort/2*, *tail/2*, *head/2*, and *element/2* (Fig. 1). Given this background knowledge, the minimal tree cost program in the hypothesis space is shown in Fig. 2b.⁴ This minimal cost program has the following tree cost:

Proposition 1 *Find duplicate minimal cost program* Let n be the list length. Then the tree cost of the minimal tree cost program is $O(n \log n)$.

Sketch proof 1 The minimal tree cost program involves first sorting the list and then passing through the list checking whether any two adjacent elements are the same. Thus the overall cost is $O(n \log n)$. \square

We generate a positive training example as follows:

1. Select a random integer k from the interval $[5, 100]$ to represent the size of the input
2. Select a random integer j from the interval $[1, k]$ to represent the duplicate element
3. Append j to the sequence $1 \dots k$ and randomly shuffle the resulting list to form s'
4. Form the atom $f(s', j)$

We generate a negative training example as follows:

³ All code and experimental data used in the experiments are available at <https://github.com/andrewcropper/mj18-metaopt>.

⁴ One could find the duplicate in time $O(n)$ using a hash table but this program is not in the hypothesis space, so could not be found by Metaopt.

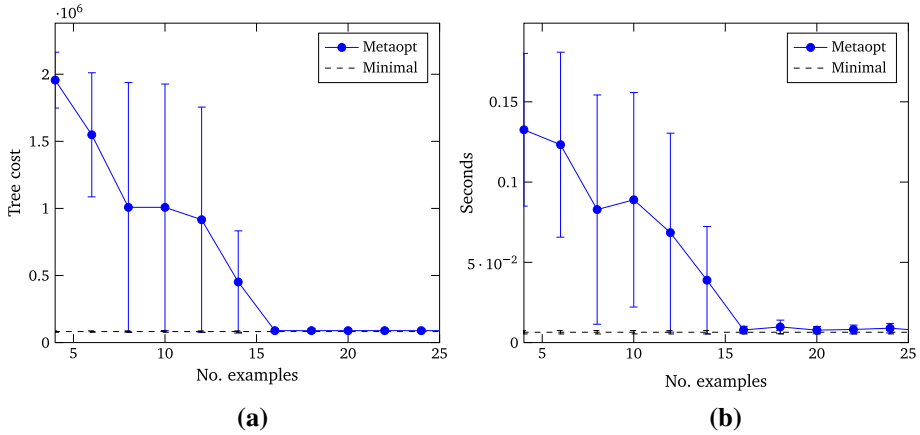


Fig. 5 **a** The median tree costs of programs learned by Metaopt when varying the number of training examples. Error bars represent the median absolute derivation. After approximately 15 examples, Metaopt converges on the minimal cost program. **b** The corresponding runtimes

1. Select a random integer k from the interval $[5, 100]$ to represent the size of the input
2. Select a random integer j from the interval $[1, k]$ to represent the false duplicate element
3. Randomly shuffle the sequence $1 \dots k$ to form s'
4. Form the atom $f(s', j)$

We generate testing examples using the same procedures but for a fixed input size $k = 1000$. *Method* Our experimental method is as follows. For each m in the set $\{4, 6, 8, \dots, 40\}$:

1. Generate m training examples, half positive and half negative
2. Generate 2000 testing examples, half positive and half negative
3. Learn a program p using the training examples with a timeout of 10 minutes.
4. Measure the tree cost and running time of p over the testing examples

We measure median tree costs and running times over 20 repetitions.

Results Figure 5a shows that Metaopt learns programs with lower costs given more training examples. After approximately 15 examples, Metaopt converges on the minimal cost program, refuting null hypothesis 1. Figure 5b shows similar results when measuring the runtimes of learned programs, and that the tree cost of a program corresponds to its time complexity.

5.2 Experiment 2: comparison with other systems

This experiment again revisits the *find duplicate* problem. The aim is to compare the tree costs and running times of programs learned by Metaopt to programs learned by Metagol and Metagol_O, where Metaopt minimises tree costs, Metagol minimises program size (Definition 4), and Metagol_O minimises resource complexity (as described in Sect. 4.3). We test the null hypothesis:

Null hypothesis 2 Metaopt cannot learn programs with lower costs and lower running times than Metagol and Metagol_O.

Materials We provide all three systems with the same background knowledge as in Experiment 1. We generate training examples in the same way as in Experiment 1. We

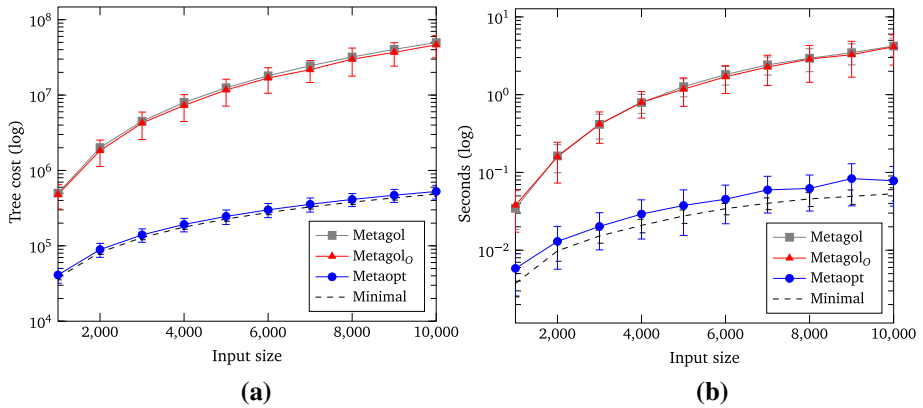


Fig. 6 **a** The median tree costs of learned *find duplicate* programs. Error bars represent the median absolute derivation. The costs of programs learned by Metaopt match those of the minimal cost program and are of order $O(n \log n)$. By contrast, the programs learned by Metagol and Metagol_O are of order $O(n^2)$. **b** The corresponding runtimes

Table 2 Examples for the *p01* string transformation problem

Input	Output
My name is John	John
My name is Bill	Bill
My name is Josh	Josh
My name is Albert	Albert
My name is Richard	Richard

generate testing examples using the same procedure but for fixed list sizes from the set $\{1000, 2000, \dots, 10,000\}$ to measure tree costs as the input grows.

Method Our experimental method is as follows:

1. Generate 20 training examples, half positive and half negative
2. Generate 100 testing examples, half positive and half negative
3. Learn a program p using the training examples with a timeout of 10 minutes.
4. Measure the tree cost and running time of p over the testing examples

We measure median tree costs and running times over 20 repetitions.

Results The log-lin plot in Fig. 6a shows that Metaopt learns programs with lower tree costs than both Metagol and Metagol_O. The log-lin plot in Fig. 6b shows similar results when measuring the runtimes of learned programs. Therefore, null hypothesis 2 is refuted both in terms tree costs and running times. Figure 2a, b show example programs learned by Metagol_O and Metaopt respectively.

5.3 Experiment 3: real-world string transformations

In Lin et al. (2014) the authors evaluate Metagol on 17 real-world string transformation problems. Table 2 shows problem *p01* where the goal is to learn a program that extracts the name from the input. This experiment explores whether Metaopt can learn minimal tree cost programs for these real-world problems.

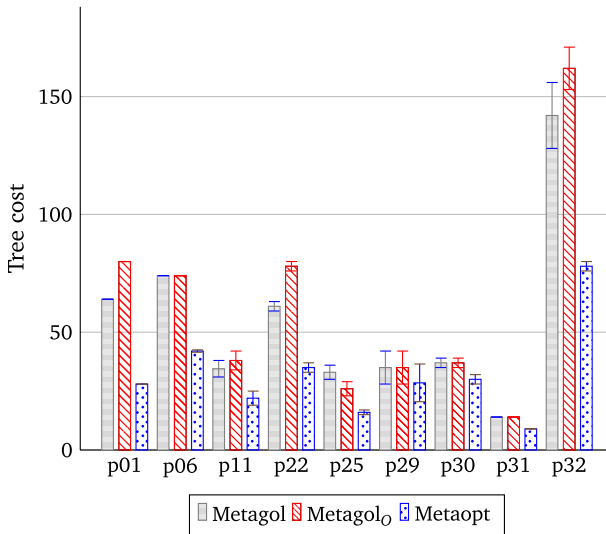


Fig. 7 Median tree costs of learned string transformation programs. Error bars represent the median absolute derivation. Metaopt learns lower tree cost programs than Metagol and Metagol_O in all cases

Materials We provide Metaopt, Metagol, and Metagol_O with the same background knowledge containing the *curry* and *chain* metarules (Table 1) and the predicates: *is_letter/1*, *not_letter/1*, *is_uppercase/1*, *not_uppercase/1*, *is_number/1*, *not_number/1*, *is_space/1*, *not_space/1*, *tail/2*, *dropLast/2*, *reverse/2*, *filter/3*, *dropWhile/3*, and *takeWhile/3*.

Method The dataset from Lin et al. (2014) contains five examples of each problem. We perform leave-two-out (keep-three-in) cross validation. We measure median program costs and running times over all trials. We set a timeout at 10 minutes.

Results Out of the 17 problems, Metagol, Metagol_O, and Metaopt learned different programs for 9 of them. Figure 7 shows the tree costs for the 9 problems, where Metaopt learns programs with lower costs in all cases, again refuting null hypothesis 2. For problem *p01*, the cost of the program learned by Metaopt (31) is half of that learned by Metagol (67) and Metagol_O (86). Figure 8 shows example programs learned by the systems for problem *p01*. Although textually more complex, the program learned by Metaopt has a lower tree cost because it successively applies the *tail/2* predicate until it reaches the first letter of the name. By contrast, Metagol learns a program which uses the *dropWhile/3* predicate to recursively check whether the head symbol is uppercase, and if not drops the head element, which requires twice the amount of work. Because Metagol_O only associates costs with dyadic predicates, it found a program which does not directly use any primitive dyadic predicates, and so has a resource cost of 0, yet is less efficient than the one found by Metaopt.

5.4 Experiment 4: robot postman strategies

In Cropper and Muggleton (2015), Metagol_O is shown to learn robot strategies with lower resource complexities than Metagol. This experiment reproduces the postman experiment from Cropper and Muggleton (2015) to show that Metaopt can simulate Metagol_O by treating resource complexity as a specific case of the cost minimisation problem, i.e. by using the resource complexity implementation described in Sect. 4.2.2. We test the null hypotheses:

Fig. 8 Programs learned by Metagol, Metagol_o, and Metaopt for the *p01* string transformation problem

Metagol (a)

```
f(A,B):-tail(A,C),f1(C,B).
f1(A,B):-dropLast(A,C),f2(C,B).
f2(A,B):-dropWhile(A,B,not_uppercase).
```

Metagol_o (b)

```
f(A,B):-f_1(A,C),f4(C,B).
f1(A,B):-f_2(A,C),f3(C,B).
f2(A,B):-filter(A,B,is_letter).
f3(A,B):-dropWhile(A,B,is_uppercase).
f4(A,B):-dropWhile(A,B,not_uppercase).
```

Metaopt (c)

```
f(A,B):-tail(A,C),f1(C,B).
f1(A,B):-f2(A,C),dropLast(C,B).
f2(A,B):-f3(A,C),f3(C,B).
f3(A,B):-tail(A,C),f4(C,B).
f4(A,B):-f5(A,C),f5(C,B).
f5(A,B):-tail(A,C),tail(C,B).
```

Null hypothesis 3 Metaopt cannot learn programs with lower resource complexities than Metagol.

Materials Imagine a humanoid robot postman learning to collect and deliver letters in a d sized one-dimensional space. In the initial state, the robot is at position 1 and n letters are to be collected. In the final state, the robot is at position 1 and n letters have been delivered to their intended destinations. The state is represented as a list of Prolog facts. The robot can perform primitive actions to transform the state: *move_right/2*, *move_left/2*, *pick_up_left/2*, *pick_up_right/2*, *drop_left/2*, *drop_right/2*, *take_letter/2*, *bag_letter/2*, and *give_letter/2*. All primitive actions have a cost of 1. The robot can also perform complex actions, which are defined in terms of primitive actions: *find_next_sender/2* and *find_next_recipient/2*, *go_to_start/2*, and *go_to_end/2*. The costs of complex actions are determined by their constituent primitive actions. The robot can take and carry a single letter from a sender using the action *take_letter/2*. Alternatively, the robot can take a letter from a sender and place it a postbag using the action *bag_letter/2*, which allows the robot to carry multiple letters. We use the *chain* and *tailrec* metarules (Table 1).

We generate a training example using the following procedure:

1. Select a random integer d from the interval $[10, 25]$ representing the number of houses.
2. Select a random integer n from the interval $[1, 5]$ representing the number of letters.
3. For each letter l , select random integers i and j from the interval $[1, d]$ representing the letter's start and end positions, such that $i \neq j$
4. Form an input state $s_1 =$

$$[\text{pos}(\text{pman}, 0), \text{energy}(0), \text{pos}(l_1, i_1), \dots, \text{pos}(l_n, i_n), \\ \text{letter}(l_1, i_1, j_1), \dots, \text{letter}(l_n, i_n, j_n)]$$

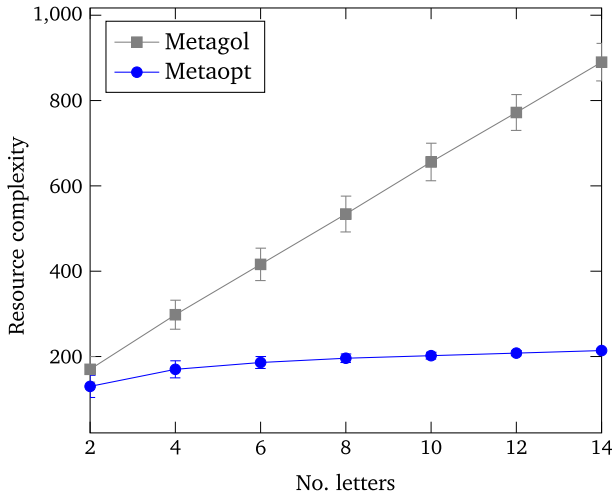


Fig. 9 Median resource complexity of learned postman strategies with varying numbers of letters for 50 places. Error bars represent the median absolute derivation

5. Form an output state $s_2 =$

$$[pos(pman, _), energy(_), pos(l_1, j_1), \dots, pos(l_n, j_n), \\ letter(l_1, i_1, j_1), \dots, letter(l_n, i_n, j_n)]$$

6. Form an example $f(s_1, s_2)$.

We generate a test example using the aforementioned procedure but with a fixed number of letters $d = 50$ and fixed number of letters n from the set $\{2, 4, \dots, 20\}$ to measure the resource complexity as n grows.

Methods Our experimental method is as follows:

1. Generate 5 positive training and 100 positive testing examples
2. Learn a program p using the training examples with a timeout of 10 minutes.
3. Measure the resource complexity and running time of p over the testing examples

We measure median resource complexities of learned strategies over 20 trials.

Results Figure 9 shows that Metaopt learns strategies with lower resource complexities than Metagol, refuting null hypothesis 3. Figure 10 shows two strategies learned by Metagol (a) and Metaopt (b), able to handle any number of houses, any number of letters, and different start/end positions for the letters. Although the strategies are equal in their textual complexity, they differ in their resource complexity. The strategy learned by Metaopt (b) has lower resource complexity because it involves the use of the postbag to store letters, whereas strategy (a) does not.

6 Conclusions and further work

We have introduced Metaopt which extends MIL to support learning minimal cost logic programs. To find minimal cost programs, Metaopt uses iterative descent, which iteratively learns lower cost programs, each time further restricting the hypothesis space. We have

Metagol (a)
<pre>f(A,B):-go_to_start(A,C),f1(C,B). f(A,B):-f1(A,C),f(C,B). f1(A,B):-find_next_recipient(A,C),f2(C,B). f1(A,B):-find_next_sender(A,C),take_letter(C,B). f2(A,B):-give_letter(A,C),go_to_start(C,B).</pre>
Metaopt (b)
<pre>f(A,B):-f1(A,C),f1(C,B). f1(A,B):-f2(A,C),go_to_start(C,B). f1(A,B):-f2(A,C),f1(C,B). f2(A,B):-find_next_recipient(A,C),give_letter(C,B). f2(A,B):-find_next_sender(A,C),bag_letter(C,B).</pre>

Fig. 10 Postman strategies learned by Metagol (a) and Metaopt (b) with resource complexities $O(n+d)$ and $O(nd)$ respectively

shown (Theorem 1) that given sufficiently large numbers of examples, Metaopt converges on minimal cost programs, and that in practice (Experiment 1), only small numbers of examples are required. To learn minimal time-complexity programs, we introduced a cost function called tree cost (Definition 16), which is based on the size of a SLD-tree at the point of which a goal is proved by a logic program. Our experiments on the *find duplicate* problem show that Metaopt learns minimal cost programs given small numbers of examples. By contrast, Metagol and Metagol_O both learn non-minimal cost programs with longer running times. Our experiments also show that Metaopt learns programs with lower costs than existing systems on some real-world string transformation problems. Finally, our experiments on learning robot strategies show that Metaopt can simulate Metagol_O and learn minimal resource complexity robot strategies by treating resource complexity as a specific case of the cost minimisation problem.

6.1 Future work

Theorem 1 shows that Metaopt learns minimal cost programs given sufficient examples. The *find duplicate* experiment (Sect. 5.1) supported this result and showed that in practice only small numbers of examples (<20) are necessary. Future work should further test this result on other domains, such as learning from visual data or learn efficient taleo-reactive programs (Nilsson 1994). Likewise, in all of our experiments, we have assumed noise-free examples, which means that a learned program must be consistent with all examples. This assumption restricts MIL from being applied to noisy problems. To address this limitation, we could relax the requirement that a program must be consistent with all examples. One method, similar to the one used by Muggleton et al. (2018), is to repeatedly learn programs from random subsets of the examples, and to then calculate confidence levels of the learned programs based on the size of the subsets and the number of repetitions.

Other complexity measures We have introduced techniques to learn minimal worst-case complexity programs. However, we are often interested in finding minimal average-case complexity programs, and future work should explore this topic. To do so, the framework in Sect. 3 would need to be adjusted to accommodate a different function to measure the cost of a program over a set of examples (Definition 3), which would effect the proof of

convergence of Metaopt (Theorem 1). In this case, it would be desirable to analyse how many examples Metaopt would need to converge on the optimal average-case program. Another promising areas of future work include investigating whether Metaopt can learn minimal space-complexity programs or minimal power consumption programs.

Program complexity analysis Metaopt uses iterative descent to continually prune the hypothesis space of programs that are less efficient than already learned ones. However, this approach is inefficient when the first found program (in the first iteration of iterative descent) has a prohibitively high cost. For instance, suppose you are learning to sort lists and that the shortest program in the hypothesis space is permutation sort. Then in the first iteration of iterative descent, Metaopt would find permutation sort, which would require $O(n!)$ time. If the examples are large, then this approach would be impractical. To overcome this issue, iterative descent could start with a low program cost bound and then iteratively relax this bound until the first program is found. Once a program has been found, iterative descent could then work as it does now and search for more efficient programs by continually restricting the hypothesis space. Alternatively, we could estimate the tree complexity of a program by approximating the SLD-tree size (Kilby and Slaney 2006).

Algorithm discovery We have used Metaopt to learn efficient programs, such as an efficient quicksort robot strategy and an efficient *find duplicate* program. However, although the learning techniques are novel, the learned programs are not, i.e. we have learned programs that we already knew about. In future work, we want to use Metaopt for *algorithm discovery*, where the goal is to learn programs that are both efficient and novel.

Open Access This article is distributed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made.

References

- Blum, L., & Blum, M. (1975). Toward a mathematical theory of inductive inference. *Information and Control*, 28(2), 125–155.
- Cropper, A., & Muggleton, Stephen H. (2015). Learning efficient logical robot strategies involving composable objects. In *IJCAI* (pp. 3423–3429). AAAI Press.
- Cropper, A., & Muggleton, S. H. (2016a). Learning higher-order logic programs through abstraction and invention. In *IJCAI* (pp. 1418–1424). IJCAI/AAAI Press.
- Cropper, A., & Muggleton, S. H. (2016b). Metagol system. <https://github.com/metagol/metagol>.
- Debray, S. K., López-García, P., Hermenegildo, M.V., & Lin, N.-W. (1997). Lower bound cost estimation for logic programs. In *Logic programming. Proceedings of the 1997 international symposium* (pp. 291–305), Port Jefferson, Long Island, NY, USA, October 13–16, 1997
- Eiter, T., Faber, W., Leone, N., Pfeifer, G., & Polleres, A. (2003). Answer set planning under action costs. *Journal of Artificial Intelligence Research*, 19, 25–71.
- Gulwani, S. (2011). Automating string processing in spreadsheets using input–output examples. In *Proceedings of the 38th ACM SIGPLAN-SIGACT symposium on principles of programming languages, POPL 2011* (pp. 317–330), Austin, TX, USA, January 26–28, 2011
- Gulwani, S., Hernández-Orallo, J., Kitzelmann, E., Muggleton, S. H., Schmid, U., & Zorn, B. G. (2015). Inductive programming meets the real world. *Communications of the ACM*, 58(11), 90–99.
- Hoffmann, J., & Nebel, B. (2001). The ff planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14, 253–302.
- Kant, E. (1983). On the efficient synthesis of efficient programs. *Artificial Intelligence*, 20(3), 253–305.
- Kaplan, S. (1988). Algorithmic complexity of logic programs. In *Logic Programming, Proceedings of the fifth international conference and symposium* (pp. 780–793), Seattle, Washington, August 15–19, 1988 (2 Volumes).

- Kilby, P., & Slaney, J. K. (2006). Sylvie Thiébaux, and Toby Walsh. Estimating search tree size. In *AAAI* (pp. 1014–1019). AAAI Press.
- Laird, J. E. (2008). Extending the soar cognitive architecture. *Frontiers in Artificial Intelligence and Applications*, 171, 224–235.
- Law, M., Russo, A., & Broda, K. (2014). Inductive learning of answer set programs. In E. Fermé & J. Leite (Eds.), *Logics in artificial intelligence* (pp. 311–325). Berlin: Springer.
- Levin, L. A. (1984). Randomness conservation inequalities; information and independence in mathematical theories. *Information and Control*, 61(1), 15–37.
- Lin, D., Dechter, E., Ellis, K., Tenenbaum, J. B., & Muggleton, S. (2014). Bias reformulation for one-shot function induction. In *ECAI, volume 263 of Frontiers in artificial intelligence and applications* (pp. 525–530). IOS Press.
- Manna, Z., & Waldinger, R. (1979). A deductive approach to program synthesis. In *IJCAI* (pp. 542–551). William Kaufmann .
- Mitchell, T. M. (1997). *Machine learning*., McGraw Hill series in computer science New York: McGraw-Hill.
- Moyle, S., & Muggleton, S. H. (1997). Learning programs in the event calculus. In N. Lavrač, & S. Džeroski, S. (Eds.), *Proceedings of the seventh inductive logic programming workshop (ILP97)*, LNAI 1297 (pp. 205–212). Berlin: Springer-Verlag.
- Muggleton, S. H., Dai, W-Z., Sammut, C., Tamaddoni-Nezhad, A., Wen, J., & Zhou, Z-H. (2018). Meta-interpretive learning from noisy images. *Machine Learning*. <https://doi.org/10.1007/s10994-018-5710-8>.
- Muggleton, S. (1995). Inverse entailment and prolog. *New Generation Computing*, 13(3&4), 245–286.
- Muggleton, S., & Feng, C. (1990). Efficient induction of logic programs. In *ALT* (pp. 368–381).
- Muggleton, S., Srinivasan, A., & Bain, M. (1992). Compression, significance, and accuracy. In D. H. Sleeman & P. Edwards (Eds.), *Proceedings of the ninth international workshop on machine learning (ML 1992)* (pp. 338–347), Aberdeen, Scotland, UK, July 1–3, 1992. Morgan Kaufmann.
- Muggleton, S. H., Lin, D., Pahlavi, N., & Tamaddoni-Nezhad, A. (2014). Meta-interpretive learning: Application to grammatical inference. *Machine Learning*, 94(1), 25–49.
- Muggleton, S. H., Lin, D., & Tamaddoni-Nezhad, A. (2015). Meta-interpretive learning of higher-order dyadic datalog: Predicate invention revisited. *Machine Learning*, 100(1), 49–73.
- Nienhuys-Cheng, S.-H., & de Wolf, R. (1997). *Foundations of inductive logic programming*. New York: Springer.
- Nilsson, N. J. (1994). Teleo-reactive programs for agent control. *Journal of Artificial Intelligence Research (JAIR)*, 1, 139–158.
- Otero, R. P. (2005). Induction of the indirect effects of actions by monotonic methods. In: S. Kramer & B. Pfahringer (Eds.), *Inductive logic programming. 15th international conference, ILP 2005. Proceedings, volume 3625 of Lecture notes in computer science* (pp. 279–294), Bonn, Germany, August 10–13, 2005. Springer.
- Pettorossi, A., & Proietti, M. (1994). Transformation of logic programs: Foundations and techniques. *The Journal of Logic Programming*, 19(20), 261–320.
- Plotkin, G. D. (1969). A note on inductive generalisation. In B. Meltzer & D. Michie (Eds.), *Machine Intelligence* (Vol. 5, pp. 153–163). Edinburgh: Edinburgh University Press.
- Plotkin, G.D. (1971). A further note on inductive generalization. In *Machine intelligence* (Vol. 6). Edinburgh: University Press.
- Puterman, M. L. (2014). *Markov decision processes: Discrete stochastic dynamic programming*. Hoboken: Wiley.
- Russell, S. J., & Norvig, P. (2010). *Artificial intelligence: A modern approach* (3rd ed.). New Jersey: Pearson.
- Shapiro, E. Y. (1983). *Algorithmic program debugging*. Cambridge: MIT Press.
- Sterling, L., & Shapiro, E. Y. (1994). *The art of Prolog—advanced programming techniques* (2nd ed.). Cambridge: MIT Press.
- Summers, P. D. (1977). A methodology for LISP program construction from examples. *Journal of ACM*, 24(1), 161–175.
- Sutton, R. S., & Barto, A. G. (1998). *Reinforcement learning—An introduction. Adaptive computation and machine learning*. Cambridge: MIT Press.
- van Otterlo, M., & Wiering, M. (2012). Reinforcement learning and Markov decision processes. In M. Wiering & M. van Otterlo (Eds.), *Reinforcement Learning* (pp. 3–42). Berlin: Springer.
- Vera, S. (1975). Induction of concepts in the predicate calculus. In *Advance papers of the fourth international joint conference on artificial intelligence* (pp. 281–287), Tbilisi, Georgia, USSR, September 3-8, 1975.
- Wielemaker, J., Schrijvers, T., Triska, M., & Lager, T. (2012). SWI-Prolog. *Theory and Practice of Logic Programming*, 12(1–2), 67–96.

- Xing, Z., Chen, Y., & Zhang, W. (2006). Optimal strips planning by maximum satisfiability and accumulative learning. In *Proceedings of the international conference on autonomous planning and scheduling (ICAPS)* (pp. 442–446).
- Yang, F., Khandelwal, P., Leonetti, M., & Stone, P. (2014). Planning in answer set programming while learning action costs for mobile robots. *AAAI spring 2014 symposium on knowledge representation and reasoning in robotics (AAAI-SSS)*.
- Zelle, J. M., & Mooney, R. J. (1993). Combining FOIL and EBG to speed-up logic programs. In *IJCAI* (pp. 1106–1113). Morgan Kaufmann.